3. Séquence numéro 3 : héritage, listes et tableaux dynamiques. 2h CM, 2h TD, 4h TP

3.1. TD

3.2. TP - suite du projet

Au cours de cette séance, nous allons voir tout d'abord au sein de notre projet un usage classique des éléments static : le pattern Singleton.

Ensuite, nous utiliserons l'héritage et enfin le polymorphisme (sur la séance prochaine) pour dégrossir un premier moteur de rendu de jeu.

Pour mémoire, l'héritage permet à une classe d'hériter des caractéristiques d'une autres classes (attributs et méthodes), et de les spécialiser. La surcharge, que nous avons déjà vu, permet à une classe héritante de redéfinir une méthode complètement. Le polymorphisme, qui utilise ces deux concepts, permet d'appeler la "bonne" méthode lorsque plusieurs méhtodes sont surchargés.

Ainsi, si l'on prend le diagramme de classe à la fin de cette séance :

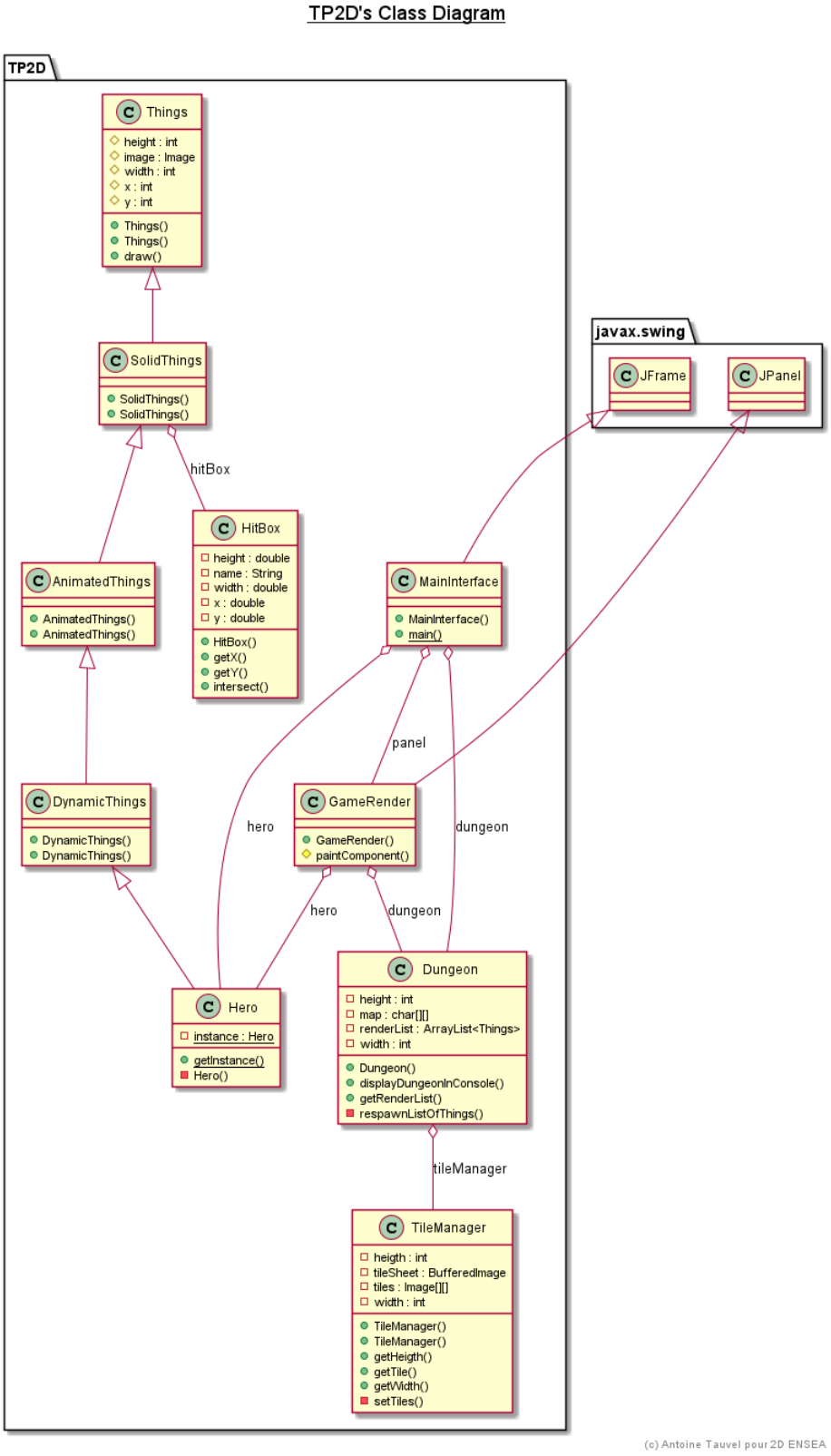

Figure 3.1.: Le diagramme de classe à la fin de la séance (si tout va bien).

On voit sur ce diagramme que les objets "Things" ont une méthode draw qui permet de les dessiner. Mais les objets de type AnimatedThings auront une autre méthodes, car ils doivent "choisir" parmi plusieurs images pour donner une impression de mouvement.

Mais démarrons par la nouvelle classe, "Hero".

3.2.1. Création de la classe héros.

Le héros va hériter de la classe DynamicThings. Il ne peut y avoir qu'un seul héros, le héros est donc un **singleton**. C'est à dire une classe ne pouvant exister qu'à un seul exemplaire. Les singletons font partis des **Design Pattern**. Un design pattern est un problème récurent en programmation pour lesquels des solutions existent qui sont agnostiques (c'est à dire qu'elle ne dépend pas du langage considéré). Le design pattern singleton est l'un des plus classiques. Voyons ce qu'en dit Wikipédia :

En génie logiciel, le singleton est un patron de conception (design pattern), appartenant à la catégorie des patrons de création, dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet. On fournira alors un accès global à celui-ci. Il s'agit d'un des patrons de création les plus simples mais les plus couramment utilisés1.

Il est utilisé lorsqu'on a besoin d'exactement un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec un seul objet qu'avec beaucoup d'objets similaires.

Principe On implémente le singleton en écrivant une classe contenant une méthode qui crée une instance uniquement s'il n'en existe pas encore. Sinon elle renvoie une référence vers l'objet qui existe déjà. Dans beaucoup de langages de type objet, il faudra veiller à ce que le constructeur de la classe soit privé, afin de s'assurer que la classe ne puisse être instanciée autrement que par la méthode de création contrôlée.

[...]

Voici une solution écrite en Java (il faut écrire un code similaire pour chaque classe singleton) :

```java
// La classe est finale, car un singleton n'est pas censé avoir d'héritier.
public final class Singleton {

    // L'utilisation du mot clé volatile, en Java version 5 et supérieure,
    // empêche les effets de bord dus aux copies locales de l'instance qui peuvent être modifiées dans le thre
    // De Java version 1.2 à 1.4, il est possible d'utiliser la classe ThreadLocal.
    private static volatile Singleton instance = null;

    // D'autres attributs, classiques et non "static".
    private String ---;
    private int zzz;

    /**
     * Constructeur de l'objet.
     */
    private Singleton() {
        // La présence d'un constructeur privé supprime le constructeur public par défaut.
        // De plus, seul le singleton peut s'instancier lui-même.
        super();
    }

    /**
     * Méthode permettant de renvoyer une instance de la classe Singleton
     * @return Retourne l'instance du singleton.
     */
    public final static Singleton getInstance() {
        //Le "Double-Checked Singleton"/"Singleton doublement vérifié" permet
        //d'éviter un appel coûteux à synchronized,
        //une fois que l'instanciation est faite.
        if (Singleton.instance == null) {
           // Le mot-clé synchronized sur ce bloc empêche toute instanciation
           // multiple même par différents "threads".
           // Il est TRES important.
           synchronized(Singleton.class) {
             if (Singleton.instance == null) {
               Singleton.instance = new Singleton();
             }
           }
        }
        return Singleton.instance;
    }

    // D'autres méthodes classiques et non "static".
    public void faire---(...) {
      ...
      this.xxx = "bonjour";
    }

    public void faireZzz(...) {
      ...
    }

}
```


Listing 5: Le Singleton selon wikipedia.

Il est à noté que nous verrons l'année prochaine en Java avancé à quoi correspond le mot-clé synchronized.

- Créez la classe Hero qui hérite de la classe DynamicThings et implémente le design pattern Singleton.
- Testez votre classe Hero en essayant de l'instancier plusieurs fois.
- Au sein de la classe DynamicThings, intégrez une logique de déplacement au sein du dongeon. La fonction `void moveIfPossible(double x, double y, Dungeon dungeon);` ne permet le déplacement que si il n'y a pas d'interractions entre les éléments solides du Dongeons et le héros. Il va donc falloir réutiliser la fonction `intersect` de la séance précédente. `Essayez votre code en faisant avancer le héros jusqu'à un mur.`

Nous allons maintenant nous attaquer au rendu graphique de notre jeu.

3.2.2. Création des constructeurs de Things et de ses héritiers

- Au sein de la classe Things, créer un constructeur ayant comme signature `public Things(int x, int y, Image image)`.
- Pour ce constructeur, les attributs width et height sont issue de l'image.
- Mettez en place un constructeur ayant cette signature pour chaque classe héritante.

3.2.3. Modification du TileManager pour la gestion des images

- Au sein de la classe TileManager, ajoutez un attribut `private Image[][] tiles` et un attribut `private BufferedImage tileSheet`.
- Écrivez une méthode `private void setTiles(int width, int height, String fileName);`. Cette méthode ouvre le fichier *fileName* à l'aide de la méthode statique `ImageIO.read()`. Référez vous à la javadoc.
- Ce début de code génère une erreur : en effet, vous ne managez pas vos exceptions. Ce concepts sera vu un peu plus tard dans le cours. Pour le moment, mettez le code dans un bloc try and catch comme ci-desssous.
- A l'aide de la méthode getSubImage, générez une sous-image pour chaque élément de la tileSheet, que vous rangerez dans le tableau.

Attention, à partir d'ici, on travaille avec trois jeux de coordonnées : parfois (x,y) désigne la position dans l'image affiché finale, parfois la coordonnée de la sous-image (dans l'exemple ci-dessous (0,1)

désigne la tuile jaune, parfois la coordonnées en tuile dans le donjon (par exemple (1,1) ici représente du sol sans mur.

```
try{
    tileSheet = ImageIO.read(new File(fileName));}
catch (Exception e){
    e.printStackTrace();
}
```

Listing 6: Extrait du code de la fonction setTiles

Figure 3.2.: Une tileSheet pas très intéressante.

- Appelez la fonction setTiles à la fin des deux constructeurs. Si *fileName* n'est pas définis, alors on appelle le setTiles avec la tileSheet d'exemple présente dans Moodle. Celle-ci à une grille de 32x32 pixels.
- Écrivez une méthode `private void setTiles(int width, int height, String fileName);`. Cette méthode ouvre le fichier *fileName* à l'aide de la méthode statique `ImageIO.read()`. Référez vous à la javadoc.
- Ce début de code génère une erreur : en effet, vous ne managez pas vos exceptions. Ce concepts sera vu un peu plus tard dans le cours. Pour le moment, mettez le code dans un bloc try and catch comme ci-desssous.
- A l'aide de la méthode getSubImage, générez une sous-image pour chaque élément de la tileSheet, que vous rangerez dans le tableau.
- Codez une méthode simple `public Image getTile (int x, int y);` permettant de récupérer une tuile du tableau tiles.

3.2.4. Modification de la classe Dungeon

- Ajoutez à la classe Dungeon une ArrayList de type privée nommée `renderList`.
- Ajoutez une méthode respawnListOfThings qui commence par vider la renderList, puis scan la map. Pour chaque case, elle génère un nouvel objet dans la liste, soit de nature Things (si elle lit un espace), ou de nature SolidThings (si elle lit un W). L'image associée doit être différente.
- Appelez la fonction respawnListOfThings à la fin du constructeur (la fonction est séparée du constructeur car on utilisera plus tard un autre constructeur.
- Ecrivez un getter permettant de récupérer la renderList.

3.2.5. Création des classes graphiques

Dans tous les frameworks d'affichage graphique, l'affichage repose sur l'héritage de classe qui intègre la gestion graphique. Dans Java Swing, deux classes vont nous intéresser :

- JFrame, qui est une fenêtre de l'OS.
- JPanel, qui est un élément d'affichage.

Création de la classe MainInterface

La classe MainInterface hérite de la classe JFrame. Une JFrame n'a pas de comportement par défaut vis à vis de l'appui sur la croix de fermeture, il faut donc coder ce comportement. Par ailleurs, on doit spécifier sa visibilité et sa taille.

Voici le code permettant de faire cela au sein du constructeur :

```
public MainInterface() throws HeadlessException {
    super();
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    //this.getContentPane().add(panel);
    this.setVisible(true);
    this.setSize(new Dimension(400,600));
}
```

Listing 7: Le constructeur de notre interface graphique. Pour le moment l'affichage du panneau GameRender est commenté

- Programmez la classe MainInterface avec son constructeur.
- Testez là dans un main situé au sein de cette classe.

Création de la classe GameRender

La classe GameRender implémente deux attributs : un héros et un Donjon. Elle hérite de la classe JPanel. Elle a deux attributs : un Dungeon et un Hero. Le constructeur les initialise avec des valeurs passées en argument.

Il est nécessaire de surcharger la méthode `protected void paintComponent(Graphics g);` qui permet de définir ce qu'il se passe lors de la création graphique de cette élément. Elle appelle la méthode draw pour chaque élément de la renderList du donjon.

A vous maintenant de coder la classe GameRender avec son constructeur et la méthode paintComponent. Ajouter un GameRender à la classe MainInterface.
Admirer votre oeuvre.

Personnalisation du donjon

Hummmm... Ce donjon ne semble pas très joli. A l'aide du site `http://opengameart.org/` ou de vos propre talent artistique, personnalisez votre donjon.